Chapter 8

As we mention throughout the book, the JSP page should, under normal circumstances, contain no business logic of any kind. The JSP page is there to present the output of some business logic operation, nothing more. Crowding the JSP page with anything else makes it harder to write the code for the page and introduces maintenance headaches down the line. In other words, the JSP page should contain only the presentation logic.

However, it is quite often the case that the presentation logic itself is very complex and would require use of straightforward Java code to implement. Java code is something we try to avoid inside the JSP pages. What we want to do is write tag-based code inside the JSP pages and write the Java code inside regular Java classes. Java-based custom tags allow us to do just that.

In Chapter 7 (Tag Libraries: The Basics), we discussed the basics of creating and working with Java-based as well as JSP-based custom tags. In this chapter, we continue talking about Java-based custom tags while looking at some of the more advanced applications.

8.1 Manipulating Tag Body

In Section 7.5 (Including Tag Body in the Tag Output), we discussed how to include the body of the tag in the output of the Java-based custom tag. To review, there are essentially two things you need to do.

- Specify `scriptless` in the `body-content` element of the TLD for the tag. This allows the page author to insert JSP content between the opening and closing elements of the tag. Remember that this JSP content is not allowed to have any scripting elements like `<% ... %>` or `<%= ... %>`.
- Invoke the output of the tag body by calling `getJspBody().invoke(null)` inside the `doTag` method of the tag handler class. Remember that this statement passes the output of the JSP content to the client, not the actual JSP code itself.

The `invoke` method takes a `Writer` as its argument. If `null` is passed to the `invoke` method, the container directs the output of the tag body to the `JspWriter` object. The server obtains a reference to this object by calling methods similar to `getJspContext().getOut()`. In other words, the statements

```
getJspBody().invoke(null);
```

and

```
getJspBody().invoke(getJspContext().getOut());
```

accomplish exactly the same result.

This construct lets us pass a different `Writer` to the `invoke` method. Using the new `Writer`, we can buffer up the output of the JSP content, extract it from the `Writer`, manipulate it, and output the new content to the client.

The following are the steps of how this can be accomplished.

1. Create an instance of a convenient `Writer`. Any class that inherits from `java.io.Writer` class is acceptable. Because the output to the client is usually HTML, which is just a string, the `java.io.StringWriter` class is the most common `Writer` to use. For example:

   ```
   StringWriter myWriter = new StringWriter();
   ```

2. Pass the newly created `StringWriter` as an argument to the `invoke` method. For example:

   ```
   getJspBody().invoke(myWriter);
   ```

   Note that now the output of the tag body is not sent to the client but buffered up inside `myWriter`.

3. Extract the buffered output from the `Writer`, modify it, and send the modified version to the client like so:

```
String modified = modify(myWriter.toString());
getJspContext().getOut().print(modified);
```

8.2 Example: HTML-Filtering Tag

There are two types of tags in HTML that affect the style of the characters as they are rendered by the browsers: physical and logical style tag types. For example, the physical style tag `<b>` always means bold, but the logical tag `<strong>` can be bold or some other browser interpretation of strong.

In this example, we create a custom tag, `filterhtml`, that allows us to see our particular browser's interpretation of these logical style tags. We accomplish this task by creating a custom tag that filters out the HTML code, converting it to regular text, thus preventing the browser from interpreting it as HTML code. We are now able to see the actual unrendered HTML code alongside its rendered version, which is convenient for demonstration of this idea.

We define a tag handler class `HtmlFilterTag`, shown in Listing 8.1, which extends the `SimpleTagSupport` class. In its `doTag` method we pass a newly created `StringWriter` to the `getJspBody().invoke(stringWriter)` method and buffer up the output of the JSP content that the `filterhtml` tag surrounds. We extract the output from the `StringWriter`, modify it using the `ServletUtilities.filter` method, shown in Listing 8.2, and send the modified output to the client. We describe the tag to the container using the TLD `csajsp-taglib-adv.tld`. The excerpt of the `htmlfilter` tag TLD description is shown in Listing 8.3. The JSP page using the `htmlfilter` tag is shown in Listing 8.4. The result is shown in Figure 8–1.

Listing 8.1 HtmlFilterTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import coreservlets.ServletUtilities;

/** Tag that replaces special HTML characters (like less than
 *  and greater than signs) with their HTML character entities.
 */
public class HtmlFilterTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
```

Listing 8.1 HtmlFilterTag.java *(continued)*

```
    // Buffer tag body's output
    StringWriter stringWriter = new StringWriter();
    getJspBody().invoke(stringWriter);

    // Filter out any special HTML characters
    // (e.g., "<" becomes "&lt;")
    String output =
      ServletUtilities.filter(stringWriter.toString());

    // Send output to the client
    JspWriter out = getJspContext().getOut();
    out.print(output);
  }
}
```

Listing 8.2 Excerpt from ServletUtilities.java

```
package coreservlets;
import javax.servlet.*;
import javax.servlet.http.*;

/** Some simple time-savers. Note that most are static methods.
 */
public class ServletUtilities {

  /** Replaces characters that have special HTML meanings
   *  with their corresponding HTML character entities.
   */
  public static String filter(String input) {
    if (!hasSpecialChars(input)) {
      return(input);
    }
    StringBuffer filtered = new StringBuffer(input.length());
    char c;
    for(int i=0; i<input.length(); i++) {
      c = input.charAt(i);
      switch(c) {
        case '<': filtered.append("&lt;"); break;
        case '>': filtered.append("&gt;"); break;
        case '"': filtered.append("&quot;"); break;
        case '&': filtered.append("&amp;"); break;
        default: filtered.append(c);
      }
    }
```

Listing 8.2 Excerpt from ServletUtilities.java *(continued)*

```
    return(filtered.toString());
  }

  private static boolean hasSpecialChars(String input) {
    boolean flag = false;
    if ((input != null) && (input.length() > 0)) {
      char c;
      for(int i=0; i<input.length(); i++) {
        c = input.charAt(i);
        switch(c) {
          case '<': flag = true; break;
          case '>': flag = true; break;
          case '"': flag = true; break;
          case '&': flag = true; break;
        }
      }
    }
    return(flag);
  }
}
```

Listing 8.3 Excerpt from csajsp-taglib-adv.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">

  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib-adv</short-name>
  <uri>http://coreservlets.com/csajsp-taglib-adv</uri>

  <tag>
    <description>
      Converts special HTML characters such as less than
      and greater than signs to their corresponding HTML
      character entities such as &lt; and &gt;.
    </description>
    <name>filterhtml</name>
    <tag-class>coreservlets.tags.HtmlFilterTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>

</taglib>
```

Listing 8.4 html-filter.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>HTML Logical Character Styles</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>HTML Logical Character Styles</H1>
Physical character styles (B, I, etc.) are rendered consistently
in different browsers. Logical character styles, however,
may be rendered differently by different browsers.
Here's how your browser renders the HTML 4.0 logical character
styles:
<P>
<TABLE BORDER=1 ALIGN="CENTER">
<TR CLASS="COLORED"><TH>Example<TH>Result

<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib-adv.tld"
           prefix="csajsp" %>
<TR>
<TD><PRE><csajsp:filterhtml>
<EM>Some emphasized text.</EM><BR>
<STRONG>Some strongly emphasized text.</STRONG><BR>
<CODE>Some code.</CODE><BR>
<SAMP>Some sample text.</SAMP><BR>
<KBD>Some keyboard text.</KBD><BR>
<DFN>A term being defined.</DFN><BR>
<VAR>A variable.</VAR><BR>
<CITE>A citation or reference.</CITE>
</csajsp:filterhtml></PRE>

<TD>
<EM>Some emphasized text.</EM><BR>
<STRONG>Some strongly emphasized text.</STRONG><BR>
<CODE>Some code.</CODE><BR>
<SAMP>Some sample text.</SAMP><BR>
<KBD>Some keyboard text.</KBD><BR>
<DFN>A term being defined.</DFN><BR>
<VAR>A variable.</VAR><BR>
<CITE>A citation or reference.</CITE>

</TABLE>
</BODY></HTML>
```

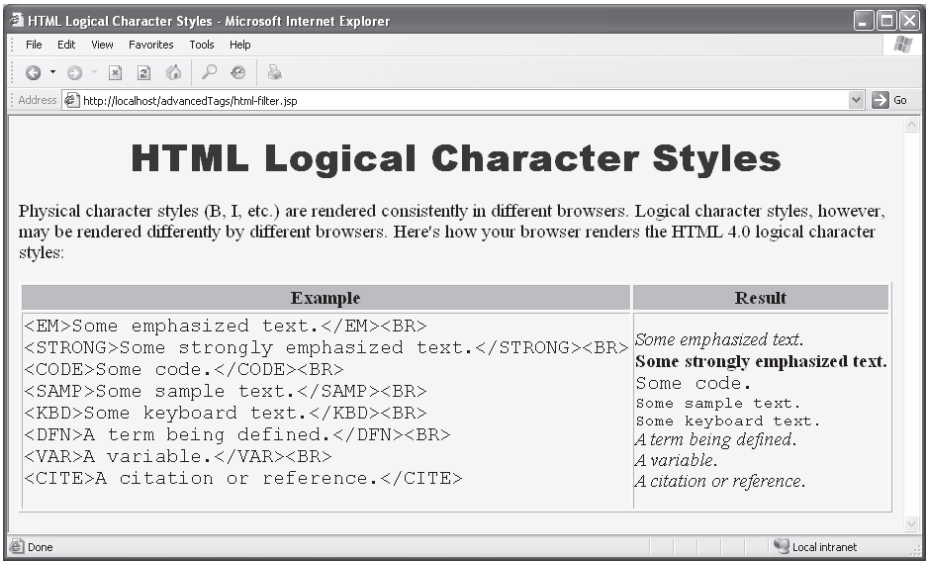

Figure 8–1 The HTML code on the left is shown unrendered by the browser because it was surrounded with the `filterhtml` tag, which converted the HTML tags to its text equivalent.

8.3 Assigning Dynamic Values to Tag Attributes

In Section 7.3 (Assigning Attributes to Tags), we discussed how to add attribute support to your custom tags. However, because of our setup we were limited to static strings as the values the JSP page was allowed to provide for those attributes. In this section, we show you how to change that so the author of the JSP page is free to pass dynamic values to your tag. In other words, we would like to be able to call our custom tag with a construct like the following:

```
<prefix:name attribute1="${bean.value}"
             attribute2="<%= bean.getValue() %>" />
```

Dynamic Attribute Values: Tag Handler Class

There is nothing that you need to do differently in the tag handler class for dynamic values than if the values for the attributes were just static strings placed into the page during development time. As far as the tag handler class is concerned, there is no difference. You still need to provide a setter method for the attribute in the form

of `setXxx(String value)`, where `Xxx` is the name of the attribute with the first character capitalized. So, if we have `attribute1` as our attribute in the tag, we would have to provide a setter method in the tag handler class like the following:

```
public void setAttribute1(String value1) {
  doSomethingWith(value1);
}
```

As before, the most common thing to do with the actual value in the setter method is to store it in a private instance variable for later use.

Dynamic Attribute Values: Tag Library Descriptor

As before, each attribute needs to be declared inside the TLD. However, because we want to allow the JSP author to specify dynamic (or runtime) expressions as values for the attributes, we must specify `rtexprvalue` to be `true`, like in the following:

```
<tag>
  <description>...</description>
  <name>mytag</name>
  <tag-class>package.TagHandler</tag-class>
  <body-content>...</body-content>
  <attribute>
    <description>...</description>
    <name>attribute1</name>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

Dynamic Attribute Values: JSP File

As before, the JSP page has to declare the tag library using the `taglib` directive. This is done in the following form:

```
<%@ taglib uri="..." prefix="..." %>
```

The usage of the tag is very much the same, except now you are able to specify JSP expressions as values for the attributes that are declared in the TLD with `rtexprvalue` of `true`. Note, however, that this does not mean that only JSP expressions are allowed to be placed as values for those attributes. Good old static string values, placed there during the authoring of the page, are still allowed. Also note that unlike the content of the tag body, the values of the attributes that accept runtime content are allowed to be JSP scripting expressions andJSP EL. In other words, the following tag, which combines JSP EL and JSP scripting expression, is perfectly legal.

```
<prefix:name attribute1="${bean.value}"
             attribute2="<%= bean.getValue() %>" />
```

8.4 Example: Simple Looping Tag

In this example, we create a simple `for` loop tag that outputs its tag body the number of times that the `count` attribute specifies. The `count` attribute is declared to accept runtime expressions, so we let its value be a JSP scripting expression that produces a random number every time the page is accessed.

Listing 8.5 shows the CoinBean.java class that produces a result of a random coin flip. Because we are allowed to use JSP EL inside the tag body, we use it to simulate a random number of coin flips inside the `for` loop by invoking its `getFlip()` method through the JSP EL notation `${coin.flip}`. This is exhibited in the simple-loop-test.jsp page, shown in Listing 8.8.

The ForTag.java class, shown in Listing 8.6, has a `setCount` method that stores the randomly produced value of the `count` attribute in the instance variable `count`. This value is then used in the `doTag` method as the number of times to loop the invocation code that outputs the tag body. Listing 8.7 shows the SimpleLoopTest.java class, which is the servlet mapped to `/simpleLoopTest` URL pattern in web.xml (Listing 8.9). The servlet creates an instance of the `CoinBean` class and stores it as a request scope attribute. The request is then dispatched to /WEB-INF/results/simple-loop-test.jsp, shown in Listing 8.8. Listing 8.10 shows an excerpt from csajsp-taglib-adv.tld, which defines the `for` tag. The result is shown in Figure 8–2.

Listing 8.5 CoinBean.java

```
package coreservlets;
import java.io.*;

/** Bean that represents a coin. It has a single method that
 *  simulates a random coin flip.
 */
public class CoinBean implements Serializable {
  public String getFlip() {
    if (Math.random() < 0.5) {
      return("Heads");
    } else {
      return("Tails");
    }
  }
}
```

Listing 8.6 ForTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Simple for loop tag. It outputs the its tag body the
 *  number of times specified by the count instance
 *  variable.
 */
public class ForTag extends SimpleTagSupport {
  private int count;
  public void setCount(int count) {
    this.count = count;
  }
  public void doTag() throws JspException, IOException {
    for(int i=0; i<count; i++) {
      getJspBody().invoke(null);
    }
  }
}
```

Listing 8.7 SimpleLoopTest.java

```
package coreservlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Simple servlet that creates a CoinBean bean, stores it
 *  in the request scope as an attribute, and forwards the
 *  request on to a JSP page. This servlet is used to
 *  demonstrate a simple looping tag.
 */
public class SimpleLoopTest extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    CoinBean coin = new CoinBean();
    request.setAttribute("coin", coin);
    String address =
      "/WEB-INF/results/simple-loop-test.jsp";
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 8.8 simple-loop-test.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Simple Loop Test</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Simple Loop Test</H1>
<P>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib-adv.tld"
           prefix="csajsp" %>
<H2>A Very Important List</H2>
<UL>
  <csajsp:for count="<%=(int)(Math.random()*10)%>">
    <LI>Blah
  </csajsp:for>
</UL>
<H2>Some Coin Flips</H2>
<UL>
  <csajsp:for count="<%=(int)(Math.random()*10)%>">
    <LI>${coin.flip}
  </csajsp:for>
</UL>
</BODY></HTML>
```

Listing 8.9 Excerpt from web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
         "http://java.sun.com/xml/ns/j2ee
          http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
  <servlet>
    <servlet-name>SimpleLoopTester</servlet-name>
    <servlet-class>coreservlets.SimpleLoopTest</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>SimpleLoopTester</servlet-name>
    <url-pattern>/simpleLoopTest</url-pattern>
  </servlet-mapping>
</web-app>
```

Listing 8.10 Excerpt from csajsp-taglib-adv.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib-adv</short-name>
  <uri>http://coreservlets.com/csajsp-taglib-adv</uri>
  <tag>
    <description>
      Loops specified number of times.
    </description>
    <name>for</name>
    <tag-class>coreservlets.tags.ForTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <description>
        Number of times to repeat body.
      </description>
      <name>count</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

Figure 8–2 Result of simple loop test. The number of times the word "blah" displays, as well as the number of simulated coin flips, is random.

8.5 Assigning Complex Objects as Values to Tag Attributes

In Section 8.3 (Assigning Dynamic Values to Tag Attributes), we showed how to set up your custom tag to accept dynamic values from the JSP page. After that setup, we are able to pass JSP EL as well as JSP scripting expressions as values of the tag attributes, like the following:

```
<prefix:name attribute1="${bean.value}"
             attribute2="<%= bean.getValue() %>" />
```

However, the value produced by `${bean.value}` was still just a string. What if you wanted to pass a `Collection` of `Orders` or some other complex object structure? What would we have to change to accommodate that?

Luckily, the answer to this question is: very little.

Complex Dynamic Attribute Values: Tag Handler Class

We still need to provide the setter for the attribute as before. However, the type of the argument in the setter would now be the complex object type instead of `String`, like the following:

```
public void setAttribute1(SomeComplexObject value1) {
  doSomethingWith(value1);
}
```

That's it! The container takes care of the rest. Again, as before, the most common thing to do with the actual value in the setter method is to store it in a private instance variable for later use.

Complex Dynamic Attribute Values: Tag Library Descriptor

The TLD stays the same as in Section 8.3 (Assigning Dynamic Values to Tag Attributes), making sure to provide the `rtexprvalue` element with a value of `true`, as follows:

```
<tag>
  <description>...</description>
  <name>mytag</name>
```

```
    <tag-class>package.TagHandler</tag-class>
    <body-content>...</body-content>
    <attribute>
      <description>...</description>
      <name>attribute1</name>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
```

Complex Dynamic Attribute Values: JSP File

As before, the JSP page has to declare the tag library using the `taglib` directive. This is done in the following form:

```
<%@ taglib uri="..." prefix="..." %>
```

The usage of the tag is very much the same as when we had dynamic values that were strings.

```
<prefix:name attribute1="${bean.value}"
             attribute2="<%= bean.getValue() %>" />
```

However, you must make sure that the type of the runtime expression is either the type declared in the setter method's argument or a subtype of that type. For example, if the setter method looks like this:

```
public void setAttribute1(SuperClass value1) {
  doSomethingWith(value1);
}
```

Then, inside the JSP page, the expression `${bean.value}` must evaluate to either an instance of `SuperClass` or an instance of any class that inherits from `SuperClass`. If the expression `${bean.value}` evaluates to anything else, the container-generated servlet code, produced as a result of this JSP page, will not compile because of the type mismatch.

8.6 Example: Table Formatting Tag

In this example, we list the three most recent swimming world records, as listed in the FINA (Fédération Internationale de Natation) database. To list the records, we employ the use of a custom table-formatting tag that lets the JSP author pass the entire record set to the tag as a two-dimensional array object.

We start off with the `ShowRecords` servlet, shown in Listing 8.11. This servlet is mapped inside `web.xml` (Listing 8.12) to the `/showRecords` URL. The servlet retrieves the records to be displayed using the `WorldRecords` class, shown in Listing 8.13. We store the records as a request scope attribute and forward the request to the `show-records.jsp` page shown in Listing 8.14. The `show-records.jsp` page utilizes the `makeTable` tag to display the records in a neatly formatted table. The tag handler class for this tag is `MakeTableTag`, which is shown in Listing 8.15. We declared the `makeTable` tag in the `csajsp-taglib-adv.tld` as shown in Listing 8.16. The result is shown in Figure 8–3.

Listing 8.11 ShowRecords.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet retrieves the records, stores them in the request
 *  scope, and forwards the request to show-records.jsp.
 */
public class ShowRecords extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    Object[][] records = WorldRecords.getRecentRecords();
    request.setAttribute("records", records);
    String address = "/WEB-INF/results/show-records.jsp";
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 8.12 Excerpt from web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
         "http://java.sun.com/xml/ns/j2ee
          http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
  <servlet>
    <servlet-name>RecordDisplayer</servlet-name>
    <servlet-class>coreservlets.ShowRecords</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>RecordDisplayer</servlet-name>
    <url-pattern>/showRecords</url-pattern>
  </servlet-mapping>
</web-app>
```

Listing 8.13 WorldRecords.java

```
package coreservlets;

/** This class simulates the retrieval of world records
 *  from the FINA database.
 */
public class WorldRecords {
  public static Object[][] getRecentRecords() {
    Object[][] records = {
      { "Event", "Name", "Time" },
      { "400 IM", "Michael Phelps", "4:08.25"},
      { "100 Br", "Lindsay Hall", "1:05.08"},
      { "200 IM", "Katie Hoff", "2:09.71"}};
    return(records);
  }
}
```

Listing 8.14 show-records.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Recent World Records</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Recent World Records</H1>
Following are the three most recent swimming
world records, as listed in the FINA database.
<P>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib-adv.tld"
           prefix="csajsp" %>
<CENTER>
<csajsp:makeTable rowItems="${records}"
                  headerClass="COLORED" />
</CENTER>
</BODY></HTML>
```

Listing 8.15 MakeTableTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Tag handler class for the makeTable tag. It builds an
 *  HTML table and outputs the records of the two-dimensional
 *  array provided as one of the attributes of the tag in
 *  the JSP page.
 */
public class MakeTableTag extends SimpleTagSupport {
  private Object[][] rowItems;
  private String headerClass;
  private String bodyClass;

  public void setRowItems(Object[][] rowItems) {
    this.rowItems = rowItems;
  }
```

Listing 8.15 MakeTableTag.java *(continued)*

```
  public void setHeaderClass(String headerClass) {
    this.headerClass = headerClass;
  }

  public void setBodyClass(String bodyClass) {
    this.bodyClass = bodyClass;
  }

  public void doTag() throws JspException, IOException {
    if (rowItems.length > 0) {
      JspContext context = getJspContext();
      JspWriter out = context.getOut();
      out.println("<TABLE BORDER=1>");
      Object[] headingRow = rowItems[0];
      printOneRow(headingRow, getStyle(headerClass), out);
      for(int i=1; i<rowItems.length; i++) {
        Object[] bodyRow = rowItems[i];
        printOneRow(bodyRow, getStyle(bodyClass), out);
      }
      out.println("</TABLE>");
    }
  }

}
 private void printOneRow(Object[] columnEntries,
                          String style,
                          JspWriter out)
      throws IOException {
    out.println("  <TR" + style + ">");
    for(int i=0; i<columnEntries.length; i++) {
      Object columnEntry = columnEntries[i];
      out.println("    <TD>" + columnEntry + "</TD>");
    }
    out.println("  </TR>");
  }

  private String getStyle(String className) {
    if (className == null) {
      return("");
    } else {
      return(" CLASS=\"" + headerClass + "\"");
    }
  }
}
```

Listing 8.16 Excerpt from csajsp-taglib-adv.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib-adv</short-name>
  <uri>http://coreservlets.com/csajsp-taglib-adv</uri>

  <tag>
    <description>
      Given an array of arrays, puts values into a table
    </description>
    <name>makeTable</name>
    <tag-class>coreservlets.tags.MakeTableTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <description>
        An array of arrays. The top-level arrays
        represents the rows, the sub-arrays represent
        the column entries.
      </description>
      <name>rowItems</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <description>
        Style sheet class name for table header.
      </description>
      <name>headerClass</name>
      <required>false</required>
    </attribute>
    <attribute>
      <description>
        Style sheet class name for table body.
      </description>
      <name>bodyClass</name>
      <required>false</required>
    </attribute>
  </tag>
</taglib>
```

Figure 8–3 Result of show-records.jsp.

8.7 Creating Looping Tags

In Section 8.1 (Manipulating Tag Body), we discussed how to manipulate the content of the tag body. That approach boils down to buffering up all of the output of the JSP content surrounded by the tag, modifying it in some way, and sending the modified version to the client. But what if we wanted to control only a small part of the tag body with the rest of it unchanged?

Consider the following typical Java looping structure. In it, we use a `for` loop to iterate through an array of strings.

```
for (int i; i < someArray.length; i++) {
  System.out.print("Object at position " + i + "is: ");
  System.out.print(someArray[i]);
}
```

The looping index variable, `i`, is what makes each iteration through the loop unique. The `for` loop construct exposes the looping index `i` to its body. However, no code outside of the `for` loop body is able to access `i` because its scope is limited to the body of the loop, delimited by the curly braces.

This construct is very useful inside a JSP page as well. In this case, the looping structure would be the custom tag. This tag would create some bean with appropriate values and pass it to the JSP content inside the body of the tag. This is done inside the `doTag` method with the use of the `tag body` scope attribute. The `tag body` scope is similar to the `request` or `application` scope in nature, except that its attributes are only visible to the tag body. Any JSP code outside of the custom tag's beginning and ending elements would not be able to access it. You use the following code inside the `doTag` method to place an object as an attribute of the `tag body` scope:

```
getJspContext().setAttribute(key, object);
```

You then use JSP EL inside the body of the custom tag to access this attribute. This is no different than accessing any other scoped bean. You just have to remember that this attribute is not available outside the confines of the tag body.

```
<prefix:custom-tag>
  some text ${someBean.someValue}
</prefix:custom-tag>
```

When creating a looping tag, it is also very common to provide the author of the JSP page with an attribute they can set, which lets them pass the name of the attribute they will later access in the tag body; that is, let them specify the string value of the `key` argument that gets passed into `getJspContext().setAttribute(key, object)`. For example:

```
<mytags:for beanKeyValue="arrayValue" iterateOver="${someArray}">
  Value is: ${arrayValue}
</mytags:for>
```

Section 8.8 (Example: ForEach Tag) shows an example of creating a `foreach` looping tag that utilizes this approach.

8.8 Example: ForEach Tag

In this example, we create a `forEach` custom tag that can iterate over arrays of any objects. The array is passed to the tag handler class through the tag's `items` attribute. Because we are interested in outputting the contents of the array to the page, we let the users of our tag (i.e., the page authors) designate a key through which they can refer to the current element in the loop iteration inside the body of the tag. This feature makes the tag very generic in nature because we are no longer dictating how the output should look. We are not sending back any HTML from the tag, just the values. The page authors are free to represent the collection of data in the array any way they choose.

To demonstrate the usage of the `forEach` tag, we create a servlet called `LoopTest`, shown in Listing 8.17. This servlet creates two arrays. The first is a one-dimensional array containing names of J2EE Web servers. The second is a two-dimensional array we used in Section 8.6 (Example: Table Formatting Tag), containing recent world swimming records from the FINA database. We use the `getRecentRecords` method of the `WorldRecords` class (Listing 8.13) to populate the two-dimensional array. Once the arrays are created, they are placed into the request scope, and the request is forwarded onto the `loop-test.jsp` page, shown in Listing 8.18. The JSP page uses the `forEach` tag to loop through the two arrays. Note that in the case of the two-dimensional array, we are able to nest the looping tags just like we would in regular Java code. Using the `var` attribute of the `forEach` tag, we assign a key that we use to refer to the current element of the array inside the body of the tag as the elements of the array are being looped over.

The tag handler class of the `forEach` tag is implemented by the `ForEachTag` class, shown in Listing 8.19. The TLD definition of the `forEach` tag is listed in `csajsp-taglib-adv.tld`, shown in Listing 8.20. The servlet declaration and mapping is shown in the excerpt of the `web.xml` file of Listing 8.21. The resultant page is shown in Figure 8–4.

Listing 8.17 LoopTest.java

```
package coreservlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** This servlet creates two arrays and stores them in the
 *  request scope as attributes and forwards the request
 *  to the loop-test.jsp page.
 */
public class LoopTest extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String[] servers =
      {"Tomcat", "Resin", "JRun", "WebLogic",
       "WebSphere", "Oracle 10g", "JBoss" };
    request.setAttribute("servers", servers);
    Object[][] records = WorldRecords.getRecentRecords();
    request.setAttribute("records", records);
    String address = "/WEB-INF/results/loop-test.jsp";
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 8.18 loop-test.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Loop Test</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Loop Test</H1>
<P>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib-adv.tld"
           prefix="csajsp" %>
<H2>Some Java-Based Servers</H2>
<UL>
  <csajsp:forEach items="${servers}" var="server">
    <LI>${server}
  </csajsp:forEach>
</UL>
<H2>Recent World Records</H2>
<TABLE BORDER=1>
<csajsp:forEach items="${records}" var="row">
  <TR>
  <csajsp:forEach items="${row}" var="col">
    <TD>${col}</TD>
  </csajsp:forEach>
  </TR>
</csajsp:forEach>
</TABLE>
</BODY></HTML>
```

Listing 8.19 ForEachTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** This class is a tag handler class for the forEach custom
 *  tag. It is able to iterate over an array of objects.
 */
```

Listing 8.19 ForEachTag.java *(continued)*

```
public class ForEachTag extends SimpleTagSupport {
  private Object[] items;
  private String attributeName;

  public void setItems(Object[] items) {
    this.items = items;
  }

  public void setVar(String attributeName) {
    this.attributeName = attributeName;
  }

  public void doTag() throws JspException, IOException {
    for(int i=0; i<items.length; i++) {
      getJspContext().setAttribute(attributeName, items[i]);
      getJspBody().invoke(null);
    }
  }
}
```

Listing 8.20 Excerpt from csajsp-taglib-adv.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib-adv</short-name>
  <uri>http://coreservlets.com/csajsp-taglib-adv</uri>

  <tag>
    <description>
      Loops down each element in an array
    </description>
    <name>forEach</name>
    <tag-class>coreservlets.tags.ForEachTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <description>
        The array of elements.
      </description>
```

Listing 8.20 Excerpt from csajsp-taglib-adv.tld *(continued)*

```
      <name>items</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <description>
        The name of the scoped variable to which
        each entry is assigned.
      </description>
      <name>var</name>
      <required>true</required>
    </attribute>
  </tag>
</tablib>
```

Listing 8.21 Excerpt from web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
         "http://java.sun.com/xml/ns/j2ee
          http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
  <servlet>
    <servlet-name>LoopTester</servlet-name>
    <servlet-class>coreservlets.LoopTest</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LoopTester</servlet-name>
    <url-pattern>/loopTest</url-pattern>
  </servlet-mapping>
</web-app>
```

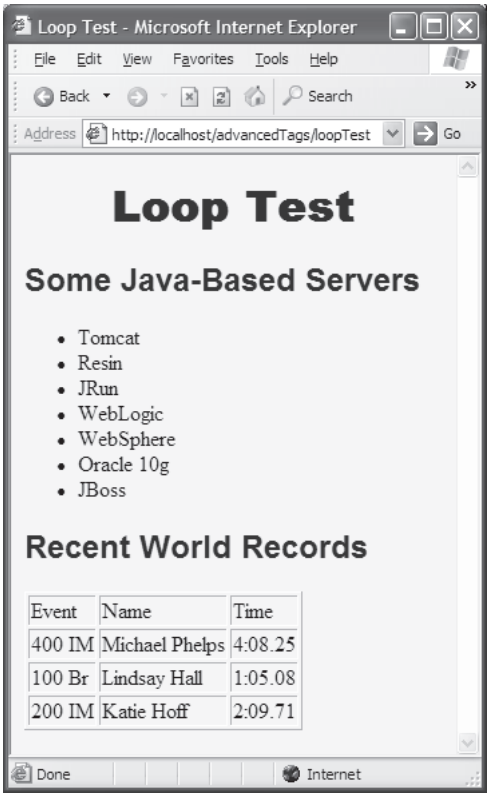

Figure 8–4 Resultant view of the page that uses the `foreach` custom tag.

8.9 Creating Expression Language Functions

In Section 7.7 (Example: Debug Tag), we showed an example of creating a `debug` tag. This simple tag surrounds some debugging information inside the JSP page. If the `debug` request parameter is present, the contents of the `debug` tag are allowed to be processed and output to the JSP page. This provides a very convenient mechanism for outputting debugging information while developing a Web page.

There is, however, one pretty significant limitation to our `debug` tag. As with all custom tags based on the `SimpleTag` API, the body of the tag is not allowed to contain any JSP scripting. The only way we can output something is through JSP EL, which requires that the object has bean-like getter methods. No doubt, we can create

such a bean in the `doTag` method of the `debug` tag handler class, but this would require us to update the tag handler class code every time we need to output some new debugging information or modify the current output. Furthermore, we would like to reuse the same `debug` tag on multiple JSP pages. Placing the information we want to see for one page into the tag handler class would require us to create multiple `debug` tags for different JSP pages. In other words, the `debug` tag would be very tightly coupled with the particular JSP page.

There are two sensible solutions to this dilemma. One would involve the usage of nested custom tags. The nested custom tag would output the debugging information specific to one JSP page. If the inner custom tag does not need to communicate with the outer custom tag, there wouldn't be anything different you would need to do to create it from what we discussed throughout Chapter 7 (Tag Libraries: The Basics). You still need to implement a tag handler class, declare the tag in some TLD, and use it inside the JSP page. The situation where the inner tag needs to communicate with the outer tag and vice versa is discussed in Section 8.11 (Handling Nested Custom Tags).

The second solution, and the topic of this section, is to use an EL function. EL functions are a new feature to the JSP specification 2.0. EL functions allow the developer to call a static method inside the JSP page, but instead of using JSP scripting to do it, which is illegal inside the tag body, EL functions allow the developer to use EL notation to invoke the function. Inside the JSP page, an EL function invocation would look like the following:

```
${prefix:someFunction(package.ClassName argument)}
```

The steps for creating an EL function are very similar to those required to create a custom Java-based tag.

1. **Create a class with a `public static` method.** This class should be placed into the `WEB-INF/classes` directory just like any servlet, filter, and so on. If this class has to come as part of a JAR file, the JAR file has to be placed inside the `WEB-INF/lib` directory. The return type of the method can be any type, but usually it is either a `String` or `void`.

   ```
   public class SomeClass {
     public static void someMethod() {...}
   }
   ```

2. **Declare the method inside a TLD file.** This step is very similar to declaring a tag inside a TLD file. The fully qualified class name and one of its methods gets mapped to some name that will be used inside the JSP page to invoke that method. However, instead of the `tag` element, which declares a tag, EL functions use the `function` element and its related elements as follows:

```
<function>
  <name>run</name>
  <function-class>somePackage.SomeClass</function-class>
  <function-signature>void someMethod()</function-signature>
</function>
```

There are a couple of things that are important to emphasize here. The value of the `function-class` element must be the fully qualified name of the Java class that implements the method. The value of the `function-signature` element should contain the signature of the method with all the specified types using their fully qualified class notation. In other words, if `someMethod` takes an argument of type `String`, the TLD must use `java.lang.String` to declare it, as follows:

```
...
<function-signature>
  void someMethod(java.lang.String)
</function-signature>
...
```

Core Warning

The values of `function-class` and `function-signature` have to contain fully qualified Java classes.

3. **Declare tag library inside the JSP page.** This step is identical to declaring a TLD that contains declarations of custom tags exclusively. Just like before you use the `taglib` directive, assign a prefix to be used throughout the JSP page. For example:

   ```
   <%@ taglib prefix="myfunc" uri="someURI" %>
   ```

4. **Use JSP EL to invoke the method.** The invocation of the method is done in the following form:

   ```
   ${myfunc:run()}
   ```

   The `myfunc` part is the prefix that comes from the `taglib` declaration mentioned in Step 3. The `run` part comes from the `name` element inside the `function` element declaration in the TLD. Note that the function name used inside the JSP page does not have to be the same as the method name inside the implementing class.

8.10 Example: Improved Debug Tag

In Section 7.7 (Example: Debug Tag), we introduced a `debug` tag. To review, the `debug` tag lets us output part of the JSP page designated as debugging information. If one of the request parameters sent to the page is `debug`, the contents of the JSP page containing the debugging information, which are surrounded by the `debug` tag, are allowed to be output to the client. If no `debug` request parameter is present, the page executes without outputting the debugging information.

However, because the debugging information is inside a custom tag, we are very limited in the type of content we are allowed to output. Namely, we are only allowed to use JSP EL. For example, we can't use JSP scripting to invoke a utility method. However, now that we know how to use JSP EL functions, we can get around this limitation.

The `debug` tag itself does not need to be changed at all. Therefore, in this example, the `DebugTag` class, shown in Listing 8.22, remains unchanged. Instead, we define a `Util` class (Listing 8.23) with a `public static` method called `information`. We map this method inside the `csajsp-taglib-adv.tld` file (Listing 8.24) to a JSP EL function named `info`. Then, we are able to use JSP EL inside the `debug.jsp` page (Listing 8.25) to invoke the `info` function as part of the `debug` tag body. Figure 8–5 shows the result of invoking the `debug.jsp` page with a number of request parameters.

Listing 8.22 DebugTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.http.*;

/** DebugTag outputs its body if the request parameter
 *  'debug' is present and skips it if it's not.
 */
public class DebugTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    PageContext context = (PageContext)getJspContext();
    HttpServletRequest request =
      (HttpServletRequest)context.getRequest();
    // Output body of tag only if debug param is present.
    if (request.getParameter("debug") != null) {
      getJspBody().invoke(null);
    }
  }
}
```

Listing 8.23 Util.java

```
package coreservlets.jsp;
import java.util.Enumeration;
import javax.servlet.http.HttpServletRequest;

/** Utility class whose method is used as an JSP EL function. */
public class Util {
  public static String information(HttpServletRequest request) {
    String result = "";
    result += "Agent Header: " + request.getHeader("User-Agent");
    result += "<BR>";
    result += "Parameters:<BR>";
    Enumeration paramNames = request.getParameterNames();
    while (paramNames.hasMoreElements()) {
       String paramName = (String) paramNames.nextElement();
       result += paramName + "<BR>";
    }
    return result;
  }
}
```

Listing 8.24 Excerpt from csajsp-tablib-adv.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib-adv</short-name>
  <uri>http://coreservlets.com/csajsp-taglib-adv</uri>

  <tag>
    <description>Conditionally outputs debug info</description>
    <name>debug</name>
    <tag-class>coreservlets.tags.DebugTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
```

Listing 8.24 Excerpt from csajsp-tablib-adv.tld *(continued)*

```
  <function>
    <name>info</name>
    <function-class>coreservlets.jsp.Util</function-class>
    <function-signature>
      java.lang.String
      information(javax.servlet.http.HttpServletRequest)
    </function-signature>
  </function>
</taglib>
```

Listing 8.25 debug.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some Hard-to-Debug Page</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Some Hard-to-Debug Page</H1>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib-adv.tld"
           prefix="csajsp" %>
Top of regular page. Blah, blah, blah.
Yadda, yadda, yadda.
<csajsp:debug>
<H2>Debug Info:</H2>
**********<BR>
-Remote Host: ${pageContext.request.remoteHost}<BR>
-Session ID: ${pageContext.session.id}<BR>
-${csajsp:info(pageContext.request)}
**********<BR>
</csajsp:debug>
<P>
Bottom of regular page. Blah, blah, blah.
Yadda, yadda, yadda.
</BODY></HTML>
```

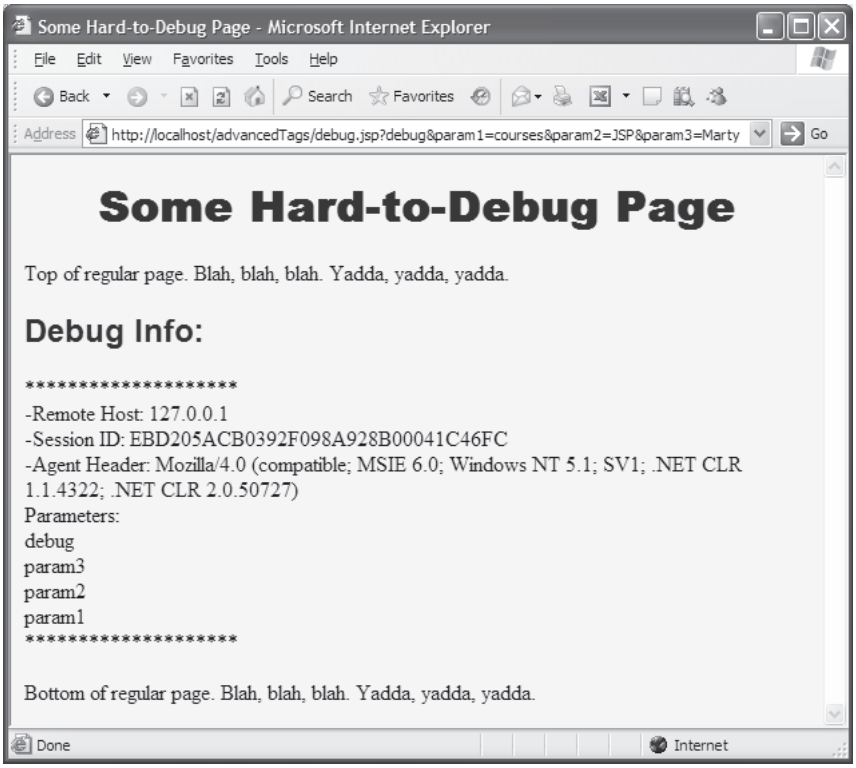

Figure 8–5 Result of invoking the `debug.jsp` page, specifying the `debug` request parameter.

8.11 Handling Nested Custom Tags

So far we have seen custom tags whose bodies contained ordinary JSP content. Of course, as discussed in Section 7.5 (Including Tag Body in the Tag Output), we take care to use only JSP EL and EL functions in the tag body. However, the tag body of a custom tag can also contain other custom tags, as follows:

```
<prefix:tagOuter>
  <prefix:tagInner>
  ...
  </prefix:tagInner>
</prefix:tagOuter>
```

Note that, just like in XML, the inner tag has to be closed before the outer tag. There is really nothing different about constructing these tags if the inner tag and outer tags don't depend on each other; that is, the inner tag can appear some other place without the presence of the outer tag and vice versa, and neither tag needs to share data with the other. However, frequently, these tags need to interact in some way. The `SimpleTag` API provides two methods that let an inner tag get a hold of the outer tag. The first method is `getParent()`. It is called on the instance of the inner tag. It returns an object of type `JspTag`, which can be cast to the type of the outer tag's handler class. If this method is called on the most outer tag in the hierarchy, it returns `null`.

Although you can keep calling the `getParent` method over and over until you either reach the tag instance you are seeking or `null`, if at the top of the tag hierarchy, the `SimpleTag` API provides another convenient method that lets you avoid repeat calls with `getParent()`. This is a static method that is provided as part of the `SimpleTagSupport` class called `findAncestorWithClass(JspTag fromTag, Class toBeMatchedClass)`. This method starts searching the hierarchy of tags from a given tag instance, `fromTag`, looking at its parent tag up until it finds a tag instance that matches the type of `toBeMatchedClass`. If the search takes it all the way to the top of the hierarchy with no results, it returns `null`. If this method finds the right instance of the tag class, it returns it as a `JspTag` instance, which needs to be cast to the proper type. There is another advantage to using the `findAncestorWithClass` method over the `getParent` method. With the `getParent` method, you need to anticipate a `ClassCastException` because you don't know the parent tag's type in advance. With `findAncestorWithClass`, however, this is not an issue. If it returns anything other than `null`, you are guaranteed that that you can successfully cast down the instance it returns using the class type that was passed in as a second argument to the `findAncestorWithClass` method.

So, if you are working inside of some inner tag, you have a way to get at the outer tag instance. What about the other way around? What method does the API provide to get at the inner tag from the outer tag? None.

If you remember our discussion about the tag life cycle from Section 7.1 (Tag Library Components), this should make perfect sense. Remember that a new instance of the tag handler class is created for every tag occurrence on the page. This rule means that after completing the instantiation of the outer tag, the inner tag instance doesn't yet exist. It would therefore be impossible to create a method that gets a hold of an instance that hasn't been created yet.

All these reasons are great, but the question still remains: How do we get the outer tags to communicate with the inner tags? The answer is to store some information in the outer tag that the inner tag can later retrieve and act on, as follows:

```
public class OuterTag extends SimpleTagSupport {
  public void setSomeValue(SomeClass arg) { ... }
  public SomeClass getSomeValue() { ... }
```

```
  public void doTag() { ... }
}

public class InnerTag extends SimpleTagSupport {
  public int doTag() throws JspException, IOException {
    OuterTag parent =
      (OuterTag)findAncestorWithClass(this, OuterTag.class);
    if (parent == null) {
      throw new JspTagException("nesting error");
    }
    SomeClass value = parent.getSomeValue();
    DoSomethingWithValue(value);
  }
  ...
}
```

Similarly, an inner tag can affect other inner tags by getting a hold of an instance of the outer tag and setting some value in it that can later be read by some other inner tag and dealt with accordingly.

8.12 Example: If-Then-Else Tag

In this example, we are going to create three custom tags: `if`, `then`, and `else`. Obviously, these tags have to work together to produce a meaningful result. The `if` tag has an attribute, `test`, that evaluates to `true` or `false`. This condition is stored in an instance variable of the `IfTag` class, shown in Listing 8.26, for which we provide the required setter method as well as the getter method. These methods are used by the nested tag handler classes.

The `then` tag gets a hold of the `if` tag instance, acquires the `test` attribute value, and allows its body to be processed if the value is `true`. The `else` tag also gets a hold of the `if` tag instance and reads the `test` attribute value, but allows its body to be processed if the value is `false`. In both `ThenTag` and `ElseTag` classes, shown in Listing 8.27 and Listing 8.28, we make sure that the result of the `getParent` method is actually an instance of the `IfTag` class. If the call to the `getParent` method does not return an instance of the `IfTag` class, a `ClassCastException` will occur. This means that the author of the JSP page made a mistake in the nesting order. We catch the `ClassCastException` and rethrow it as a `JspTagException` with the appropriate message. Likewise, the `getParent` method will return `null` if either the `then` or the `else` tags are outside of the `if` tag. Again, in such a case, we throw a `JspTagException`.

Note that in our example we use the `getParent` method and not the `findAncestorWithClass` method. In the case of the `if-then-else` construct, it is not appropriate to have other custom tags in the tag hierarchy between either the `then` or `else` tags and the `if` tag. If we used the `findAncestorWithClass` method, it would return an instance of the `if` tag even if it wasn't a direct parent of either the `then` or `else` tags. Therefore, its use in our example would be inappropriate.

The declaration of these tags is inside the csajsp-taglib-adv.tld file shown in Listing 8.29. We use the if-test.jsp file, shown in Listing 8.30, to demonstrate their use. The result is shown in Figure 8–6.

Listing 8.26 IfTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Tag handler class for the if tag. It relies on the
 *  required 'test' attribute and stores the evaluated
 *  condition in the test instance variable to be later
 *  accessed by the ThenTag.java and ElseTag.java.
 */
public class IfTag extends SimpleTagSupport {
  private boolean test;

  public void setTest(boolean test) {
    this.test = test;
  }

  public boolean getTest() {
    return(test);
  }

  public void doTag() throws JspException, IOException {
    getJspBody().invoke(null);
  }
}
```

Listing 8.27 ThenTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Tag handler class for the then tag. It gets a hold of
 *  the IfTag instance and processes its body if the value
 *  test attribute of the IfTag is true. It also throws
 *  a JspTagException if the parent of this tag is anything
 *  other than an instance of the IfTag class.
 */
public class ThenTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    // Get parent tag (if tag)
    IfTag ifTag = null;
    try {
      ifTag = (IfTag)getParent();
    }
    catch (ClassCastException cce) {
      String msg = "Error: 'then' must be inside 'if'.";
      throw new JspTagException(msg);
    }
    if (ifTag != null) {
      // Decide whether to output body of then
      if (ifTag.getTest()) {
        getJspBody().invoke(null);
      }
    } else {
      String msg = "Error: 'then' must be inside 'if'.";
      throw new JspTagException(msg);
    }
  }
}
```

Listing 8.28 ElseTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
```

Listing 8.28 ElseTag.java *(continued)*

```
/** Tag handler class for the else tag. It gets a hold of
 *  the IfTag instance and processes its body if the value
 *  test attribute of the IfTag is false. It also throws
 *  a JspTagException if the parent of this tag is anything
 *  other than an instance of the IfTag class.
 */
public class ElseTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    // Get parent tag (if tag)
    IfTag ifTag = null;
    try {
      ifTag = (IfTag)getParent();
    }
    catch (ClassCastException cce) {
      String msg = "Error: 'else' must be inside 'if'.";
      throw new JspTagException(msg);
    }
    if (ifTag != null) {
      // Decide whether to output body of else
      if (!ifTag.getTest()) {
        getJspBody().invoke(null);
      }
    } else {
      String msg = "Error: 'else' must be inside 'if'.";
      throw new JspTagException(msg);
    }
  }
}
```

Listing 8.29 Excerpt from csajsp-taglib-adv.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib-adv</short-name>
  <uri>http://coreservlets.com/csajsp-taglib-adv</uri>
```

Listing 8.29 Excerpt from csajsp-taglib-adv.tld *(continued)*

```
  <tag>
    <description>If tag</description>
    <name>if</name>
    <tag-class>coreservlets.tags.IfTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <description>Condition of the if</description>
      <name>test</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
  <tag>
    <description>Then tag (goes with If tag)</description>
    <name>then</name>
    <tag-class>coreservlets.tags.ThenTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
  <tag>
    <description>Else tag (goes with If tag)</description>
    <name>else</name>
    <tag-class>coreservlets.tags.ElseTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>
```

Listing 8.30 if-test.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>If Test</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>If Test</H1>
<P>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib-adv.tld"
           prefix="csajsp" %>
```

Listing 8.30 if-test.jsp *(continued)*

```
<H2>SSL Usage</H2>
<csajsp:if
    test="${pageContext.request.protocol==https}">
  <csajsp:then>Using SSL.</csajsp:then>
  <csajsp:else>Not using SSL.</csajsp:else>
</csajsp:if>
<H2>Coin Tosses</H2>
<UL>
  <csajsp:for count="5">
    <LI><csajsp:if test="<%=Math.random()<0.5%>">
          <csajsp:then>Heads</csajsp:then>
          <csajsp:else>Tails</csajsp:else>
        </csajsp:if>
  </csajsp:for>
</UL>
</BODY></HTML>
```

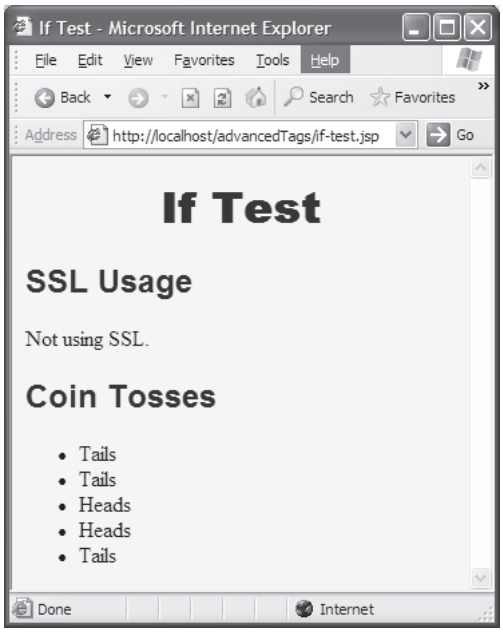

Figure 8–6 Result of if-test.jsp.